

AFRL-RI-RS-TM-2008-18
In House Technical Memorandum
April 2008



HARDWARE BASED FUNCTION LEVEL MANDATORY ACCESS CONTROL FOR MEMORY STRUCTURES

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TM-2008-18 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/

/s/

DUANE GILMOUR, Chief
Computing Technology Applications Br.

JAMES A. COLLINS, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

| | | | | | |
|---|-------------------------|--------------------------------|---|--|---|
| REPORT DOCUMENTATION PAGE | | | | <i>Form Approved</i> OMB No. 0704-0188 | |
| <small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.</small> | | | | | |
| PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS. | | | | | |
| 1. REPORT DATE (DD-MM-YYYY) APR 2008 | | 2. REPORT TYPE Final | | 3. DATES COVERED (From - To) May 07 – Sep 07 | |
| 4. TITLE AND SUBTITLE HARDWARE BASED FUNCTION LEVEL MANDATORY ACCESS CONTROL FOR MEMORY STRUCTURES | | | | 5a. CONTRACT NUMBER In-House | |
| | | | | 5b. GRANT NUMBER | |
| | | | | 5c. PROGRAM ELEMENT NUMBER N/A | |
| 6. AUTHOR(S) Lok Kwong Yan | | | | 5d. PROJECT NUMBER 230B | |
| | | | | 5e. TASK NUMBER LY | |
| | | | | 5f. WORK UNIT NUMBER SA | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AFRL/RITB 525 Brooks Rd Rome NY 13441-4505 | | | | 8. PERFORMING ORGANIZATION REPORT NUMBER | |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RITB 525 Brooks Rd Rome NY 13441-4505 | | | | 10. SPONSOR/MONITOR'S ACRONYM(S) | |
| | | | | 11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TM-2008-18 | |
| 12. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-2295 | | | | | |
| 13. SUPPLEMENTARY NOTES | | | | | |
| 14. ABSTRACT This report presents the results of the mini-grant research project. It explores a possible explanation on why buffer overflows, format strings and other memory related vulnerabilities are still prevalent today. It is argued that this can be attributed to the required level of user interaction to apply today's solutions. Therefore, the researched solution was a hardware based instruction level mandatory access control mechanism that will be enabled by default whenever a user obtains a new computer with such a processor. It also presents the reasoning behind why instruction level is more desirable than function level access control mechanisms, which was the original theory. The design and proof of concept demonstration as well as difficulties in achieving the desired proof are also presented. | | | | | |
| 15. SUBJECT TERMS Buffer overflows, mandatory access control, no execution bit, memory tagging | | | | | |
| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT UU | 18. NUMBER OF PAGES 22 | 19a. NAME OF RESPONSIBLE PERSON Lok Kwong Yan |
| a. REPORT U | b. ABSTRACT U | c. THIS PAGE U | | | 19b. TELEPHONE NUMBER (Include area code) N/A |

Table of Contents

| | | |
|-----|--|----|
| 1 | Introduction | 1 |
| 2 | Background..... | 1 |
| 3 | Related Work..... | 3 |
| 3.1 | Safe Languages | 4 |
| 3.2 | Compilers: StackGuard and CRED | 4 |
| 3.3 | Libraries and Applications | 5 |
| 3.4 | Hybrid Approaches | 6 |
| 3.5 | OS Level: PaX and Address Space Layout Randomization | 6 |
| 3.6 | Hardware: Dynamic Information Flow Tracking..... | 6 |
| 3.7 | Hardware: Secure Bit2 | 7 |
| 3.8 | Categorization of Past Solutions | 7 |
| 3.9 | Secure Architectures | 8 |
| 4 | Design..... | 10 |
| 4.1 | Failure of Function Level Access Control | 10 |
| 4.2 | Instruction Level Access Control | 10 |
| 5 | Proof of Concept..... | 13 |
| 5.1 | Bochs..... | 13 |
| 5.2 | Assembly Instruction Changes..... | 14 |
| 5.3 | Memory Mapping..... | 14 |
| 6 | Results | 15 |
| 6.1 | Conclusion..... | 15 |
| 7 | References | 17 |

List of Figures

| | |
|---|----|
| Figure 1: Diagram illustrating the range of user intervention to implement various buffer overflow solutions..... | 8 |
| Figure 2: Simple Policy | 13 |
| Figure 3: Illustration of Executable Tagging..... | 15 |

1 Introduction

The original intention of this mini-grant was to research the applicability and effectiveness of a hardware based mandatory access control mechanism operating at the functional level. It was envisioned that a mandatory access control mechanism that operated at the function level, i.e. a system that enforces read, write, and execute permissions for memory blocks based on the current executing function, is a good balance between required changes to a processor's architecture, required user interaction and the type of memory problems it can mitigate. This type of protection will provide an additional layer of defense against memory based threats such as buffer overflows. The hardware mechanism was expected to make the processor more robust against buffer overflows independent of the underlying operating system.

The results were different. We learned that access control at the functional level is inadequate for mitigating memory based threats. Also, integrating function level protections would require drastic changes to a processor's architecture, a highly undesirable trait. Finer-grained control at the byte or word level is better for both protection and ease of implementation.

This paper will present the results of the mini-grant research. It will explore a possible explanation on why buffer overflows, format strings and other memory related vulnerabilities are still so prevalent. We argue that the most successful solutions are ones that can be integrated and applied without requiring any *end-user* intervention. Following this concept, the most successful solution should be a hardware based one that can be integrated into commodity processors with some operating system support. Like the No eXecution (NX) bit technology, it should be easily integrated into current processor architectures at low cost, while remaining backwards compatible with older software. But unlike the NX bit technology, its use should not be discretionary upon the ring 3 user's desire, but should have at least some mechanisms that are mandatory and uncontrollable even at the ring 0 privileged operating system level.

The paper is organized as follows. Section 2 will introduce the underlying reason why buffer overflow and related memory based vulnerabilities exist. Section 3 will enumerate some of today's buffer overflow solutions. The idea of categorizing solutions based on the requisite amount of end user interaction is also described in this section. Section 4 will discuss design considerations. Section 5 will discuss the proof of concept. Results and conclusions will be presented in Section 6.

2 Background

The first great computer infection, the Morris Worm, made use of a buffer overflow exploit in the *fingerd* application as its method of propagation [1]. This was back in 1988. Given the extensive damage of the worm, one would think that a solution to inoculate the world's computers from such a sickness would be developed immediately. However, in 2001, thirteen years after the Morris Worm, the Code-Red and CodeRedII worms used another buffer overflow that cost \$2.6 billion to recover from [2]. This is a testimony to the ineffectiveness of the developed solutions during that thirteen year period and the

overall prevalence of this attack scheme. In fact, even in 2007 the number of buffer overflow vulnerabilities is expected to be an all time high [3].

After reviewing the available literature, we have concluded that the problem is not the lack of solutions, but of a pervasive one that can be enabled and adapted easily by large populations and that actually solves the underlying problem.

For analysis purposes, we define memory based vulnerabilities as vulnerabilities that arise from the way that system memory is organized and used. They can be binned into four large groups: un-initialized data, memory leaks, data disclosure and data corruption. Un-initialized data problems arise when an application programmer fails to initialize objects to fail-safe defaults. Memory leaks occur when applications allocate memory but lose access to them; although, the operating system can retrieve the memory after the application completes. Data disclosure problems occur when memory that should not be readable by an application is. Data corruption threats are the ones where memory data is overwritten when it was not part of the application designed to do so.

Unlike un-initialized data and memory leak vulnerabilities, data disclosure and corruption vulnerabilities cannot be fully attributed to programmer error. Although programmers can properly bounds check their buffers, the underlying problem exists because the CPU has ubiquitous access to all of memory, i.e., memory is world read, write and executable by the CPU. On the other hand, we consider uninitialized data and memory leak problems software issues because processors do not have built in constructs to allocate memory on the object level, this is all done in software.

The common problem amongst the four of them is the task of managing and protecting memory is left to the operating system's memory management unit (MMU). When a process requests memory space from the operating system, the MMU rounds the requested space to the nearest page and allocates those pages to the application. The operating system then keeps track of the pages and performs access control on each page to ensure that other processes cannot access the same address space unless explicitly permitted. On the other hand, the CPU operates on instructions and bytes of memory.

This imbalance in the unit of protection in the operating system and the CPU, as well as reliance on the operating system to offer access control for memory pages, is what enables many of the memory based vulnerabilities. For example, buffer overflows are successful because a process can overwrite all in-band (contiguous with reference to the accessed variable) data as long as the data is within the page or even across pages provided the pages are contiguous. This is true even though the write should be restricted to only within the buffer itself. Unfortunately though, current processor designs and operating systems can only control access on pages and not objects.

Upon review, we have concluded that there are three possible solutions to this problem: increase the granularity of the operating system's memory access controls to memory objects, implement access controls in hardware or a hybrid of the two. We believe that a hybrid approach that implements mandatory access controls in hardware and provides discretionary access control mechanisms for operating system use is the most desirable solution. This approach has the most potential to affect a great number of users and provides foundational support for a defense in depth strategy.

3 Related Work

As previously mentioned, many solutions have been proposed to close off the buffer overflow attack vector. We believe that buffer overflow vulnerabilities are still high impact items, not because the solutions are technically ineffective, but practically ineffective. They either solve the wrong problem, that is prevent execution of user supplied code instead of preventing the overflow in the first place, or because they require too much user sophistication or interaction, which restricts its use.

In the following subsections, a few solutions and why we believe they are ineffective are outlined. To understand the level of user interaction required, each will be placed into the following categories ordered from high to low user-interaction: language, compiler, library, application, operating system, and hardware. Who the *user* is undoubtedly changes from solution to solution but the *end-user* will always remain the same.

To keep things simple, we define the *user* as the user of the solution, who will change depending on the solution, and the *end-user* as the normal everyday person who simply uses computers and is not cognizant of vulnerabilities, patches or anything related to the topic. To illustrate, the *user* in language based solutions is the developer, in application based solutions is the administrator, and the *end-user* in both of these remains the same. Given this distinction, the further away the *user* and *end-user* are in terms of computer systems know-how, the less desirable the solution. In other words, the most desirable solution is one where all end-users can adopt readily.

Safe languages are considered the highest level of required *user* interaction because the *user* in this case must be the skilled programmer who takes code written in an unsafe language like C, and translates it into a safe language like Java. Such a task is extremely difficult, costly and sometimes impossible when it comes to legacy code. Furthermore, even if all of this is possible, the new version of the application must still be tested to ensure backwards compatibility and the old version upgraded. Given the large number of potential applications, requiring all developers to re-implement their software and administrators to upgrade existing software, makes this solution almost impossible to be widely adopted.

Compiler based solutions are better than language based solutions, since all that is required is a recompilation of available source code, although it is ineffective for situations when source code is not available. It also suffers from the installation and distribution issue of safe languages. The *user* in this case does not need to be a developer, just someone with the know-how of building new packages and then installing them.

Library and application level solutions are much better since now the user doesn't even need to know any languages, all they need to do is install the new library or application. Therefore, the *user* is an administrator.

Operating system (OS) solutions are similar to application level solutions, but are slightly better because all computing systems require operating systems which normally come pre-installed with new computers. Unfortunately though, this will require that all operating systems must be updated to support this new feature. Since operating systems these days comes with automatic patches and updates, the *user* for this solution can actually be the *end-user* as long as the OS vendor supplies the patch.

Hardware based solutions require the least user intervention and are the most pervasive, because changing the CPU or obtaining one through a new computer will mean that the solution is built in and it doesn't matter which operating system is used. The *user* can be the *end-user*, but that is only when a new system is purchased. For older systems, the *user* must be an administrator or technician type since they must install and configure the new hardware. Unfortunately, hardware only solutions are difficult to develop. A more reasonable approach, and the approach taken in this project, is to develop a hardware solution that requires operating system support. This is still valuable since in both cases the *user* can potentially be the *end-user*.

3.1 Safe Languages

Since buffer overflow problems are primarily associated with C and C++, and more specifically their use of unsafe C library functions like `strcpy`, `memcpy`, `printf` and etc., using safer languages like Java and C# is one possible solution. In fact, most modern languages conduct bounds checking on all arrays, which would stop buffer overflows before they can occur. The biggest problem with these solutions though, is that they are not completely compatible with C and C++. What this means is that developers will have to completely reengineer their codes to adapt to the paradigms of the new language. The problem is exacerbated when legacy code is involved. In such situations, it might not even be possible to port the software to a new language because the algorithmic and implementation specifics have been lost with time.

Fortunately though, there are safer languages that are almost completely compatible with C and C++. One example is Cyclone, a C variant [4]. It introduces the concept of fat pointers, which are pointers that contain size information. By doing so, all the *user* needs to do is to change their current buffer pointers to fat pointers and bounds checking will be enforced, making the task of re-implementing software much simpler. Unfortunately though, in order for this type of solution to be effective, all software developers must jump on the band wagon and use the new language. There is nothing the *end-user* can do.

3.2 Compilers: StackGuard and CRED

StackGuard is probably one of the most well known stack smashing protection schemes available. It is a modified version of the gcc compiler that automatically inserts “canaries” into the stack before a function is called. After the function completes, and before it returns to the return address on the stack, the canary value is checked with a stored version. If the values do not match, then it means that the canary has been illegally overwritten, and the stack is corrupted [5].

CRED (C Range Error Detection) is an extension to the GNU compiler that was developed at Stanford University [6]. It relies on replacing every out-of-bounds pointer value with the address of a special OOB (out-of-bounds) object created for that value. The authors showed that it is an effective and viable solution. They ran CRED on 20 open source applications ranging in complexity from approximately 400 to 600,000 lines of code and all test cases compiled and ran successfully. Unfortunately, slowdown for these test cases range from no effect to 12 times depending on the prevalence of buffers in the application.

In addition, seven of the 20 test cases were known to have buffer overflow vulnerabilities. CRED successfully detected the overflow attempts when known exploits were run against the applications. CRED also successfully detected overflow attempts for twenty additional test cases covering stack, heap and data segment overflow vulnerabilities. This is in sharp contrast to the 50% success rate of “ProPolice, the best of the tools evaluated by Wilander and Kamkar” [6].

Even with the success of CRED, the same practicality problem as with safe languages still exists. This solution, like other compiler based approaches, requires that each vulnerable program be recompiled, which is still much better than reimplementation, in order for the protection to take effect. This means that the source code for each application must be available to the *user* that wishes to make use of CRED’s protection.

The source code requirement is not such a huge problem for the growing number of open source projects, but it still requires the *user* to have the know-how of obtaining the source, recompiling it, and then install the latest build. What this means is that the *end-users* will have to wait until a new build of the software by the developers.

3.3 Libraries and Applications

At the next level of influence and required user interaction is the shared library. Applications are also placed into this category because in both cases, the *user* will have to be an administrator who will have to install the new libraries and/or applications. This is also the point where the *end-user* might actually be able to implement the solutions. Or in other words, the *user* and *end-user* has a good chance of being the same person.

Libsafe is an example of secure libraries [7]. It intercepts certain unsafe calls, calculates the maximum allowed size of the buffer based on the stack frame address, and then calls the safer bounded variant such as *strncpy*. Although this solution has proven to be effective against some stack overflows, it still has some flaws. One problem is it does not prevent the overflow itself, in-band data within the stack frame can still be overwritten. Only the control flow data is protected. Like other library based approaches, it is only applicable when the application is dynamically linked. It will not be effective for statically linked software or for user defined functions.

Program Shepherd is a system built on top of Runtime Introspection and Optimization, a dynamic optimizer application. It seeks to stop malicious code executions by using the concepts of restricted code origins, restricted control transfers, and un-circumventable sandboxing [8]. When an application is run under this solution, the loader must first determine if the block of instructions is trusted in accordance with a security policy and the origins of the code, e.g. executable file from the disk or dynamically generated code, and tags them as executable. This first step is known as restricted code origins. Restricted control transfers refers to the restriction of jumps and branches from one block of memory to another only if it is allowed in the security policy, e.g. if the target of the branch is not tagged as executable, then control transfer should be restricted. Finally, un-circumventable sandboxing is used to ensure that all implemented security checks must be done at all times. We believe that this type of solution is going down the right direction, in terms of enforcing the execute permissions, but like other approaches,

it only targets the execution of malicious code and not the overwriting of data in the first place.

3.4 Hybrid Approaches

While Program Shepherd monitors program control flows closely, Dynamic Access Control monitors program data that might be indicative of an attack, even ones that do not alter control flow [9]. The dangers of these types of attacks are demonstrated in [10]. Dynamic Access Control requires support at both the hardware and micro-architecture level. The compiler identifies program regions where data should not be modified as per program semantics. If there is an attempt to modify this data at runtime, the hardware detects the attack. Again, since this answer to the buffer overflow problem requires a compiler addition, recompilation and source code availability are both necessary. Average performance degradation after optimizations is 14.1%. There is also an associated memory overhead.

3.5 OS Level: PaX and Address Space Layout Randomization

PaX is a Linux kernel patch written by The PaX Team whose principal author chooses to remain anonymous. PaX's main avenues of defense are to mark data as non-executable and take advantage of address space layout randomization (ASLR) [11]. By default, PaX marks the memory that holds the stack, heap, anonymous memory mappings, and any section not specifically marked as executable in an ELF file, non-executable. This prevents the standard stack-smashing attack since shell code stored in the buffer on the stack will be marked as non-executable. PaX randomizes the location of the stack, heap, loaded libraries, and executable binaries thereby greatly reducing the likelihood of success for attacks that rely on hardcoded addresses, such as a standard ret-libC attack. This protection, when combined with a hardware protection scheme such as the NX bit (discussed later) provides a powerful defense in depth solution.

Being a patch to the Linux kernel means that the target demographic that uses this software will likely be adept at computer-related issues. These kinds of users would also be capable of applying many of the other techniques and is normally not the victim of memory based vulnerabilities. For this reason, PaX may not have as much to contribute to solving buffer overflow issues as some other solutions that can be applied to Windows-based platforms, for which most exploits are written for and whose users are traditionally less tech savvy. There is no reason why the same techniques cannot be integrated into Windows-based products though.

It should also be noted that a successful attack scheme on PaX has been published in Phrack [12]. It directly calls the dynamic linker's symbol resolution procedure to get around the ASLR aspect of PaX and uses a traditional ret-libC exploit from there. This supports the idea that single solutions are inadequate.

3.6 Hardware: Dynamic Information Flow Tracking

DIFT (Dynamic Information Flow Tracking) is a hardware-based tainting mechanism [13]. It offers two different security policies depending on the amount of protection needed and overhead willing to be incurred. Data from I/O channels is considered

spurious and is marked as such. This data is stored in an extra bit and is propagated to other memory locations as it interacts with other pieces of data. DIFT makes additions to a standard processor to add the extra bits and proposes various schemes to minimize memory and performance overhead. The processor then checks the taint bits before executing an instruction or committing to a branch.

On some architectures, standard integer arithmetic instructions are indistinguishable from pointer arithmetic instructions. This poses a problem for taint bit propagation. Also, in the case of properly coded program first bounds checking a piece of data and then combining it with a trusted pointer, a perfectly safe piece of code could be marked as tainted. In this case, the operating system taint module could be invoked many times, resulting in a hefty overhead. There is also a format string attack that takes advantage of a little known *printf* feature that can bypass DIFT [14].

3.7 Hardware: Secure Bit2

Secure Bit2 also extends every memory location by one bit which is used to add semantic meaning to each word of memory [15]. This bit is moved along with its associated word by memory manipulating instructions. Words in buffers between processes get their secure bit set while all others mark the secure bit at the destination register or memory location. CALL, RET and JMP instructions check the secure bit and if set, generate an interrupt or fault signal. Modifications are required at the kernel level to set the secure bit when passing a buffer across domains. Since the address validation is done in hardware, there is little performance overhead. Memory overhead is related to the total size of memory. An additional bit is needed for each word.

This solution is closer to the type that has a chance to be implemented widely. There is minimal user intervention required and it should be immune to the standard stack-smashing attack scheme. However, there is still the possibility of overwriting in-band data stored on the stack. Such an event could cause unintended and incorrect program behavior and potentially could be used to form an attack itself. As previously mentioned, it has been shown that attacks on non-control based data are dangerous as well.

3.8 Categorization of Past Solutions

In Figure 1, the continuum of user involvement in taking advantage of the previously mentioned solutions is given. At the uppermost portion of the figure are safe languages like Java and Cyclone that require a great amount of work on the user's part. Obviously, as more labor is required of the user, the tasks become inherently more complex and require a more tech savvy user. Past a certain point there is little utility in such a solution because a user of this level of expertise is likely to have other protections installed on his/her computer and is less likely to download or run malicious software.

The goal for our solution is to have it exist at the tip of the arrow in Figure 1, representing a minimal level of user involvement and thus, skill. It is only this type of solution that can hope to be successful on a large scale. Requiring any more work or skill on the user's behalf would result in a solution that may fix the problem, but is unlikely to ever help the *end-user*.

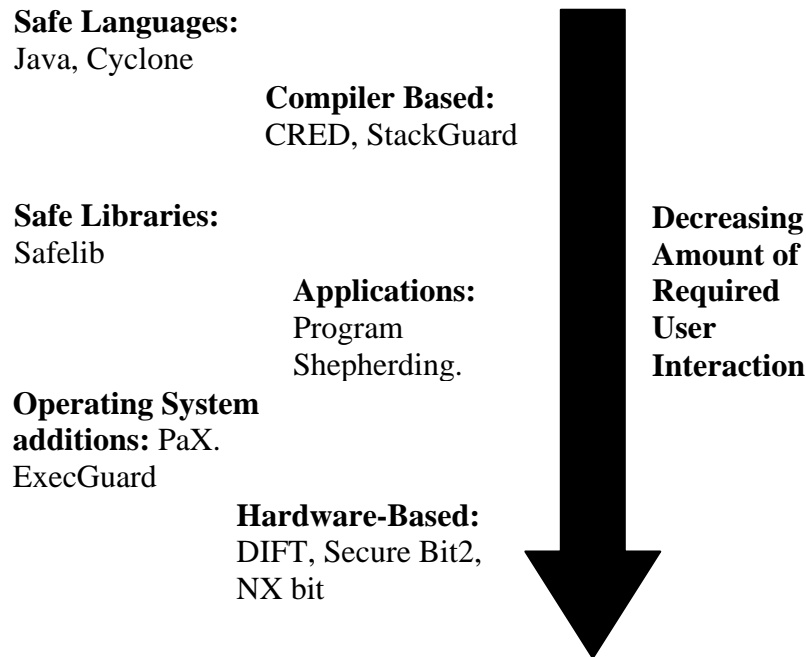


Figure 1: Diagram illustrating the range of user intervention to implement various buffer overflow solutions.

3.9 Secure Architectures

Dating all the way back to 1959, there has been an effort to create more secure processors by enforcing safety precautions at the hardware level. Capability-based architectures refer to computer systems that access data using an address that refers to both the memory object itself and a set of access rights that govern how that data can be used. For an excellent reference of past such architectures see [16].

First shown in the early 1960s, the Burroughs family of processors incorporated some very sophisticated features for their time [17]. Originally, the B5000 used a 1 bit tag as part of its 32 bit word. The B6000 expanded it to 3 bits and moved it outside the word. It differentiates data from code and control words and is even used to indicate type (such as single and double precision floating point). The hardware enforced security mechanism makes it impossible to execute data as code or to interpret code as data.

Released in 1980, the IBM System/38 sought to be a totally object-oriented architecture [18]. The System/38 featured 40 bit words consisting of 32 data bits, 7 bit ECC, and a 1 bit tag. The tag bit is set whenever the data bits contain a pointer while all other words in memory have their tag bits cleared. These tag bits cannot be accessed by the instruction interface and cannot be set by the user. Instead, they are manipulated by instructions that use microcode to build the pointers and maintain the integrity of the tag bits. User modification of the pointer results in its tag bits to be cleared making it invalid for addressing purposes.

Introduced in the year after the IBM System/38, the design and layout of the chip set for the Intel iAPX 432 took over 100 man-years [16]. Memory references are performed using 32 bit long access descriptors (ADs) that specify the actual address and access rights to an object. A procedure can only address and manipulate the ADs that are within

its execution environment. The access rights specify whether the possessor of the AD can read from or write to the object or delete the AD itself. Unfortunately, the iAPX 432 was doomed by performance problems and an overzealous marketing campaign.

A novel computer architecture that is still around today is used by Unisys Mainframes [19]. The ISA tags each word of memory to indicate how the data stored there can be used. All data references are done through descriptors generated by the hardware and operating system using instructions unavailable to ordinary user code. Every memory reference is checked for a valid descriptor and that the reference is within appropriate bounds. Programs that are running are not given privilege to descriptors that hold their own code or that of another program. Furthermore, code and data are kept separate eliminating any adjacency between buffers and areas containing executable code.

These types of ideas were brought into the mainstream when AMD began to use an extra bit, the No eXecute bit (NX), to mark pages of memory as non-executable. Intel's version of AMD's NX bit is called the Execute Disable Bit [20]. The capability of the processor to take advantage of this sort of functionality can be queried by the operating system that is running. When activated by setting the bit IA32_EFER.NXE, memory pages can be marked as not being executable. This is done by adjusting bit 63 in the corresponding page table entry for that page of memory. If the protection is running and an instruction fetch to a linear address that translates to a physical address in a memory page that has the execute disable bit set, a page fault exception will be generated. This sort of protection is very close to what is desired in protecting memory from memory based vulnerabilities: there is no effort required of the user other than having a processor with the ability, it incurs very little memory or performance overhead, and is backwards compatible with existing code.

To allow for that backwards compatibility, Intel decided to give the host OS the ability to turn this protection on or off. Windows XP Service Pack 2 and Windows 2003 Service Pack 1 contain patches to take advantage of this hardware feature by using what Microsoft calls Data Execution Prevention (DEP). Shortly after their debut, exploits began to be posted that easily sidestepped the mechanism [21]. In order to bypass DEP, a ret-libC style attack can be used to jump to a section of system code marked as executable that can then further be exploited to disable DEP and return into shell code stored in the original buffer. If there was no way that a process could disable NX support at runtime, this exploit would not work.

It is this particular fact that makes it seem that a mandatory access control mechanism that works at the hardware level would be more successful. Protection schemes at the hardware level should not be allowed to be circumvented by the OS. While it is understandable that some security tradeoffs may need to be made to allow for backwards compatibility, some basic security components should not.

4 Design

This solution borrows the ideas from current approaches and implements them at a level that has the most impact and requires the least *user* and *end-user* know-how. This can be achieved by implementing a Mandatory Access Control mechanism within the processor hardware.

By implementing the solution in hardware, all future processors can potentially have additional safe guards that can be used by the most people, like the NX bit. As support becomes more widespread, more abstract components of the CPU could take advantage of this type of protection.

There were numerous design considerations but we finalized on the same tagging architecture that operates on memory objects as that found in the original Burroughs system and others. Through our research, we have found that enabling access control at the function level required too much re-engineering of the processor's architecture.

4.1 Failure of Function Level Access Control

The largest impediment to a function level access control design was that current processors are largely function-agnostic. The lack of support for function level operations makes a function level approach extremely expensive to develop. The first problem we discovered was the existence of both branch and CALL instructions. While it is desirable to utilize the CALL instruction when available, it is not an architectural requirement. Users are allowed to directly utilize the branch instruction to jump to a predefined start of the next function. This meant that it was impossible, with current processor architectures, to confidently infer the beginning and end of a function. Thus function level access control would require a complete redesign and implementation of a different instruction invalidating all of today's software.

In addition to this debilitating result, we also realized that even if a processor with the strict call-only instruction set was available, it would still have to be redesigned to include a storage area for keeping track of all the different functions in all of the different processes that are currently operating. While it is possible to incorporate such tables into current memory hierarchies by designating certain memory areas as processor only, similar to the Linux kernel's kernel address space, it is still too difficult. To realize this approach, the processor must maintain a table for each running process and ensure that there is enough space for as many tables as there could be processes. Given that the table would require the *number of processes x number of threads / process x number of functions / thread x size of table / function* bytes where the number of processes and threads are not necessarily bounded, it would add an additional requirement on the virtual memory management unit to ensure that these tables are paged as necessary. It was deemed that this required too much behind the scenes coordination.

4.2 Instruction Level Access Control

Given the desire to keep the solution as simple as possible, we determined that enforcing access control at the instruction level would be perfect. Although it is true that instructions are even lower level than functions, it does not mean that the design and

implementation is more difficult. In fact, because processors naturally operate instructions, it is actually considerably easier. The only required changes are to provide a tag for each byte of memory and change each processing instruction to enforce a simple policy based on each byte's tag.

We concluded that tagging each byte of memory is the desired approach. This was chosen in favor of tagging each word, since memory architectures are byte addressable, even though most compilers use word boundaries. Although it does mean that there will be considerable storage overhead, it is compatible with virtual memory architectures. The only change needed is to ensure that the memory management unit reserves a portion of memory for this purpose. The tag can then be stored in this portion and is easily accessible through a simple address mask like current virtual memory architectures. This portion of memory can also be paged and loaded like any other memory locations as well. It is a significantly simpler feat than that of the function level approach, since this does not require the processor to know about running processes or functions.

We determined that one bit per byte is sufficient to provide an evolved version of the no execution functionality. Unlike current techniques though, the bit will be used to mark individual bytes as executable instead of whole pages. The advantage of marking bytes is to ensure that there is no slack space where an attacker can inject their own code that would be automatically executable because it was placed into an executable page. This approach requires an operating system level patch to ensure the loader marks each instruction byte as executable and all others within the page as non-executable. As discussed above, operating system patches are largely automated, which makes this a desirable solution. Hopefully, all the *end-user* has to do is purchase a new computer.

We also determined that two bits per byte are sufficient to prevent stack smashing and similar buffer based attacks as well. The first bit will be used as the executable flag while the second bit is used as the vector flag. The vector flag will be set if the byte is part of a vector object, i.e. array, and unset if it is a scalar. Given this distinction the processor can ensure that instructions that start writing in one type cannot continue writing to another type. This approach has its drawbacks though as the processor must be designed to be cognizant of write loops and it does not prevent overwriting from one vector variable into another vector if they are contiguous. One method to get around this weakness is to ensure that compilers separate vectors with a scalar in the middle, but even this approach is still not robust against multi-dimension arrays where the vectors must be contiguous for alignment purposes.

Perhaps a better use of this additional bit is for the processor to mark certain bytes of memory as processor only. This will ensure that the saved stack pointer and return addresses that are part of the function pre-amble can only be written to by the CALL and RET instructions. The only drawback to this approach is the wasted bit for all of the bytes which are not used solely by the processor, which is the majority by far.

We determined that the best use of the two bits is to utilize all four combinations of scalar/vector and execute/no-execute. Since only instructions are executable, and instructions are scalar objects, executable vectors should be an invalid combination. This combination can be used to represent processor only bytes. Although it does mean that a

default tag for uninitialized bytes is unavailable. Also, uninitialized bytes would have to have the same bit combination as scalar non-executable.

Given the above tagging schemes, we then developed a simple policy to enforce the scheme. The policy is shown in Figure 2.

5 Proof of Concept

To determine whether or not the above mentioned technique was feasible, a proof of concept was developed. Due to resource constraints the proof of concept did not fully show that the solution would mitigate common memory based vulnerabilities like buffer overflows, but it did show that the concept is sound.

5.1 Bochs

To implement our protection scheme, we needed a platform on which to make additions to the standard CPU hardware. During this project, we reviewed three processor

| Tag or Datatype | Policy |
|----------------------------|--|
| 00 (non-executable scalar) | MOV-like instructions must be stateful. If first write is scalar then this write is not allowed. |
| 01 (executable scalar) | Branch and call instructions can only jump to memory locations with this tag. It will fault otherwise. |
| 10 (non-executable vector) | MOV-like instructions must be stateful. If first write is vector then this write is allowed, else its not. |
| 11 (processor only byte) | CALL and RET instructions have write access. CALL sets this tag and RET unsets it. All other instructions only have read access. |

Figure 2: Simple Policy

simulators: OpenRISC 1000 Simulator, SimpleScalar and Bochs [22][23][24]. During our trials, we found that both OpenRISC and SimpleScalar were immature and lack adequate documentation. OpenRISC simulator has not been updated since May 2006 and SimpleScalar since August 2004. Unlike other documented users, we were unable to get either simulator up and running within the short time we had. Thus, we had to resort to Bochs, even though Bochs implemented the more complex x86 instruction set.

Hosted on SourceForge, Bochs is an open-source IA-32 emulator written in C++ that aims to emulate the Intel x86 CPU, common I/O devices, and features a custom BIOS. Bochs can be configured to emulate a host of different CPUs including: 386, 486, Pentium / PentiumII / PentiumIII / Pentium4, or x86-64 CPU including support for MMX, SSE and 3DNow! instructions. It is capable of running on a multitude of different host OSes and is capable of running Linux, DOS, Windows 95/98, Windows NT/2000/XP, or Windows Vista as guests OSes.

For our deployment of Bochs, we used the default configuration which compiles it to emulate a Pentium processor. Due to the extremely poor performance of the emulator, RedHat 6.0 was used as the guest OS. The guest OS was configured to include the most basic modules and the gcc compiler for building test applications.

5.2 Assembly Instruction Changes

To illustrate the feasibility of the solution, some of the assembly instructions in Bochs were changed to reflect the policy. The ADD, SUB, MUL, DIV, MOV, store instructions were altered to propagate the datatype of the source to the destination. Then the jump, branch and CALL instructions were implemented to ensure that they can only jump to memory locations that are tagged as executable, else it would log an error message to simulate a trap. Notice that this does not completely implement the above mentioned policy as there was not enough time to reengineer a state machine into the MOV instruction.

Also we decided that implementing a specialized instruction for the operating system's executable loader to tag memory locations as executable is not necessary at this point. Although it will be required when a full solution is designed, it is useless at this point because rewriting an operating system kernel to take advantage of it is a non-trivial task. Instead, we utilized a special trigger for setting and unsetting memory locations as executable.

Upon review of all available instructions, we found that the MOV instruction will always trap if used with the 0xFFFFFFFF and 0xFFFFFFFFE memory addresses under normal operation. We took advantage of this and used 0xFFFFFFFF as the trigger for setting the executable flag and 0xFFFFFFFFE as the trigger for unsetting the flag. To enable this, the MOV instruction was implemented so it maintains some state. When the MOV instruction's memory address is 0xFFFFFFFF the state is noted. The next time the MOV instruction is called, the memory address is treated as the start address for tagging. The third time the MOV instruction is called, the memory address is treated as the end address for tagging. This allows the user to tag a range of addresses as executable, similar to what the loader would have to do. All subsequent MOV instructions behave normally. Similarly the 0xFFFFFFFFE MOV instruction removed the executable flag.

5.3 Memory Mapping

In this proof of concept implementation, the tagging information was not integrated into the memory management unit. Instead of storing all of the datatype information in main memory as described above, a completely separate memory space was created within Bochs to hold the data. This was done to reduce the time required to ensure that the memory management unit is properly designed and implemented to be completely backwards compatible. The summer student who worked on this project will be conducting additional research into the memory management unit, the necessary changes, and performance effects for his master's thesis.

6 Results

As shown in the Figure 3, the specialized MOV instructions for setting and unsetting the executable flag for memory locations are fully functional. This confirms that the tagging architecture itself is feasible. Although it does not show that it is feasible when integrated as part of the standard memory address space and therefore observing all virtual memory properties. This must be confirmed at a later date.

Also, it was confirmed that the instructions indeed propagated the tagging information, which is a desired effect. Finally it was also confirmed that when the implemented policy was violated, e.g. when the JMP instruction tried to jump to a memory location that was not marked as executable, an error message was logged, simulating a trap.

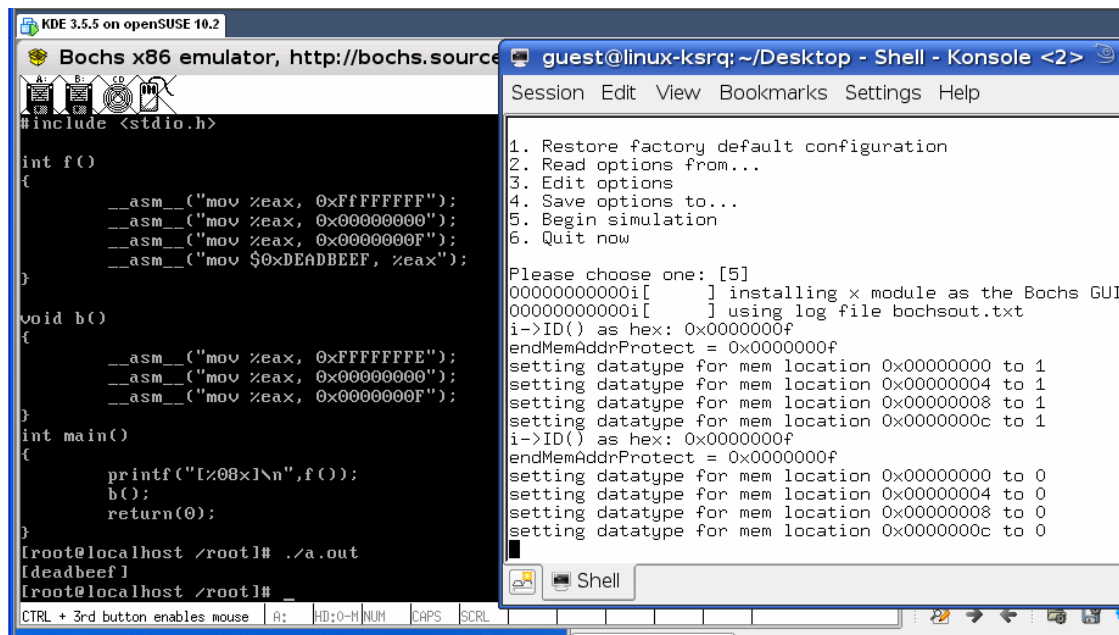


Figure 3: Illustration of Executable Tagging

Performance overhead measurements were not obtained during this proof of concept demonstration. This is due to the lack of a well defined set of metrics and measurement methodologies. Although, it is expected that performance degradation will be minimal if the memory management unit can ensure that the tagging data is loaded into the processor's cache at the same time as memory fetches.

6.1 Conclusion

All of the results show that the proposed approach is feasible to a certain extent. Although it was not possible to complete a proof of concept demonstration for a full fledged solution, the intermediate results are promising. This is especially true when we take into account the fact that the changes we made to the emulated processor did not break any of the software that runs on it. It was still completely backwards compatible. Although this might change once a true trap is utilized instead of a simple error log, this

simple proof of concept did show that the addition of tagging information for memory objects maintains backwards compatibility.

Additional work is required to continue the development of the policy and a proof of concept demonstration that incorporates the full policy as well as full functionality. Furthermore, the proof of concept should include an operating system that is capable of leveraging this new paradigm and whose loader can actually set and unset the memory locations as executable. Only in this way can we confirm that the *end-user* and *user* are the same for this solution.

Finally, during our investigation of possible tagging schemes, we have also developed schemes that utilize more than two bits. This is largely to fully cover the read and write privileges of each byte of memory. The current solution is only robust against simple buffer overflows and mitigates code execution threats. In the more advanced schemes, additional bits will be utilized to represent specific data types such as *int*, *char*, *pointer* etc. We envisioned that the additional types can help make data corruption more difficult if we ensured that you can only overwrite *ints* with *ints* or *instructions* with *instructions*. In the future, we would like to verify that there is added value in these types and also develop the policies to further mitigate data corruption and disclosure threats. We would also like to show the positive effects of utilizing the proposed solution in conjunction with other low required-user-interaction, such as OS level, solutions for a defense in depth approach.

7 References

- [1] Orman, H. "The Morris Worm: A Fifteen-Year Perspective." *IEEE Security & Privacy Magazine*, 1, 5 (Sept. – Oct. 2003), 35-43.
- [2] Moore, D., Shannon, C., and claffy, k. "Code-Red: a case study on the spread and victims of an internet worm." In *Proceedings of the 2nd ACM SIGCOMM Workshop on internet Measurment* (Marseille, France, November 06 - 08, 2002). IMW '02. ACM Press, New York, NY, 273-284. <http://doi.acm.org/10.1145/637201.637244>
- [3] National Vulnerability Database. Accessed 14 August 2007. <http://www.nvd.nist.gov/>
- [4] Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J. and Wang, Y. "Cyclone: A safe dialect of C." In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, 275-288. <http://www.cs.cornell.edu/Projects/cyclone/papers/cyclone-safety.pdf>
- [5] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In *Proceedings of the 7th USENIX Security Conference* (San Antonio, TX, Jan 1998). 63-78.
- [6] Ruwase, O. and Lam, M. A practical dynamic buffer overflow detector. In *Proc. 11th Annual Network and Distributed System Security Symposium*, Feb 2004.
- [7] Ghory, Z. "Protecting Systems with Libsafe." *Security Focus*. August 20, 2001. <http://www.securityfocus.com/infocus/1412>
- [8] Kiriansky, V., Bruening, D., and Amarasinghe, S. P. "Secure Execution via Program Shepherding." In *Proceedings of the 11th USENIX Security Symposium* (August 05 - 09, 2002). D. Boneh, Ed. USENIX Association, Berkeley, CA, 191-206.
- [9] Zhang, K., Zhang, T., and Pande, S. "Memory Protection through Dynamic Access Control." In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (December 09 - 13, 2006). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 123-134. DOI=<http://dx.doi.org/10.1109/MICRO.2006.33>
- [10] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. "Non-control-data attacks are realistic threats." In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD, July 31 - August 05, 2005). USENIX Association, Berkeley, CA, 12-12.
- [11] Silberman, P., and Johnson, R. "A Comparison of Buffer Overflow Prevention Implementations and Weaknesses." I-Defense, 1875 Campus Commons Dr. Suite 210 Reston, VA 20191, <http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf>

- [12] Nergal. "The advanced return-into-lib(c) exploits: PaX case study." Phrack Magazine 58(4), December 2001.
<http://www.phrack.org/issues.html?issue=58&id=4#article>
- [13] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. "Secure program execution via dynamic information flow tracking." In *Proceedings of the 11th international Conference on Architectural Support For Programming Languages and Operating Systems* (Boston, MA, USA, October 07 - 13, 2004). ASPLOS-XI. ACM Press, New York, NY, 85-96.
<http://doi.acm.org/10.1145/1024393.1024404>
- [14] Dalton, M., Kannan H., and Kozyrakis, C. "Deconstructing Hardware Architectures for Security." In *the 5th Annual Workshop on Duplicating, Deconstructing, and Debunking*, Boston, MA, June 2006.
- [15] Piromsopa, K., and Enbody, R. "Secure Bit2: Transparent, Hardware Buffer-Overflow Protection." Technical Reports #MSU-CSE-05-9, Department of Computer Science and Engineering, Michigan State University (2005).
- [16] Levy, H. M. *Capability-Based Computer Systems*. Butterworth-Heinemann.
- [17] Mayer, A. J. "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?" *SIGARCH Comput. Archit. News* 10, 4 (Jun. 1982), 3-10.
<http://doi.acm.org/10.1145/641542.641543>
- [18] Houdek, M. E., Soltis, F. G., and Hoffman, R. L. "IBM System/38 support for capability-based addressing." In *Proceedings of the 8th Annual Symposium on Computer Architecture* (Minneapolis, Minnesota, United States, May 12 - 14, 1981). International Symposium on Computer Architecture. IEEE Computer Society Press, Los Alamitos, CA, 341-348.
- [19] Unisys Mainframes: Secure MCP Systems – Secure by Design. Accessed 2 September 2007.
http://www.unisys.com/products/mainframes/security/secure_mcp_systems/secure_by_design.htm
- [20] "Execute Disable Bit Functionality Blocks Malware Code Execution." Intel Website. Accessed 11 September 2007. http://cache-www.intel.com/cd/00/00/14/93/149307_149307.pdf
- [21] "Bypassing Windows Hardware-Enforced Data Execution Prevention." Nologin.org. Accessed 11 September 2007. <http://uninformed.org/?v=2&a=4&t=pdf>
- [22] OpenRISC 1000 (or1k) Simulator. Accessed 24 September 2007.
<http://www.opencores.org/projects/or1k/>
- [23] SimpleScalar. Accessed 24 September 2007.
<http://www.cs.wisc.edu/~mscalar/simplescalar.html>
- [24] Bochs: The Open Source IA-32 Emulation Project. Accessed 24 September 2007.
<http://Bochs.sourceforge.net/>